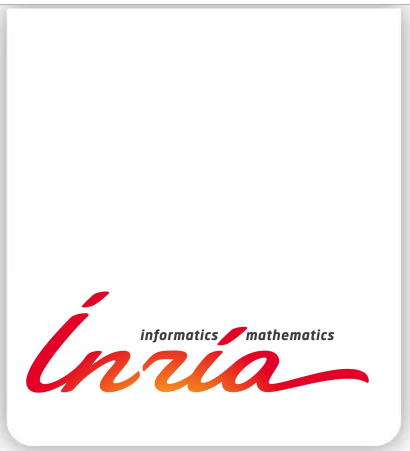
A ProActive Backend for ABS: from Modelling to Deployment

Ludovic Henrio, Justine Rochas

RESEARCH REPORT
N° 8596
September 2014
Project-Team Scale

ISSN 0249-6399 ISRN INRIA/RR--8596--FR+ENG



A ProActive Backend for ABS: from Modelling to Deployment

Ludovic Henrio, Justine Rochas

Project-Team Scale

Research Report n° 8596 — September 2014 — 18 pages

Abstract: ABS is an object-oriented modeling language that is based on a concurrent object group model, derived itself from the active object model. Its goal is to describe distributed and concurrent applications in order to verify their properties and make them safer. Thanks to the ABS Tool Suite, ABS programs can be translated into the Java programming language (among others), and executed in the JVM. This paper presents a new ABS backend that translates ABS programs into ProActive programs. ProActive is a well known active object Java library that provides support for distribution of applications across clusters or grids. The benefit of this work is to be able to easily distribute ABS programs, so that ABS models can also be experimented in a large scale setting. Our contribution includes the ProActive backend itself, the complete description of our translation strategy, and a realistic experiment that shows the benefits of the ProActive backend.

Key-words: active objects, translation, ProActive, ABS

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Un Support de Traduction ProActive pour ABS : de la Modélisation au Déploiement

Résumé : ABS est un langage de modélisation orienté objet qui est basé sur un modèle à groupe d'objets, dérivé lui-même du modèle à objets actifs. Son objectif est de décrire les applications concurrentes et distribuées dans le but de vérifier leurs propriétés et de les rendre plus sûres. Grâce à l'ABS Tool Suite, les programmes ABS peuvent être traduits dans le language Java (ainsi que dans d'autres), et être exécuté sur la JVM. Ce document présente un nouveau support de traduction qui traduit des programmes ABS en programmes ProActive. ProActive est une librairie à objets actifs Java qui intègre complètement la distribution de ses applications sur des clusters ou des grilles. La valeur ajoutée principale de ce traducteur est de pouvoir distribuer facilement des programmes ABS, pour que les modèles ABS soient aussi exécutables dans de larges environnements. Nos contributions incluent : le support de traduction ProActive, la description complète de notre technique de taduction, et une expérience réalistique qui montre les bénéfices du support de traduction ProActive.

Mots-clés : objets actifs, traduction, ProActive, ABS

1 Introduction

Writing distributed and concurrent applications can be challenging. In distributed environments, the absence of shared memory makes information sharing more difficult. In concurrent environments, information sharing is easy but might lead to inconsistent state if shared variables are not manipulated with caution. A set of languages and tools have been developed to handle those two programming challenges. Among them, the active object programming model [1] has showed to be convenient to handle concurrency and distribution at the same time. Indeed, active objects decouple method calls from method execution, and by this mean enforce sequential processing on the target side. In practice, this model is also adapted to distribution because active objects do not share memory and behave independently from each other.

There exists different implementations of the active object model. In the ASP language [2], and in its Java implementation, called ProActive [3], strict sequential execution of requests is guaranteed per active object. In other implementations, like Creol [4], requests might release temporarily the execution thread to allow another request to progress, this behavior being called cooperative scheduling. The active object model has also been applied to object groups, for example in JCoBox [5] and ABS [6]. Specifically, ABS is an object-oriented language that targets application modelling. It provides verification tools that help designing safe distributed and concurrent systems. The ABS Tool Suite [7] provides a frontend compiler and several backend translators into various programming languages. In particular, the Java backend translates ABS programs into Java programs. however, to our knowledge, there is no readily backend that offers the possibility to run the program in a distributed way. On the other hand, ProActive comes in a standard Java library and provides support for deploying active objects on clusters and grids. Thus, we present in this paper a solution to distribute ABS programs using the ProActive library. Starting from the initial Java backend, we propose the ProActive backend in order to produce a code that is easily deployable on thousands of machines. The ProActive backend makes possible to experiment ABS programs in practical settings, in addition to verify them with the existing tools. Our contribution can be summarized as the following.

- We provide the full implementation of the ProActive backend, as well as the facilities we developed for it.
- We explain in details our strategy to translate ABS into ProActive.
- We informally show that the produced behavior follows the semantics of ABS.
- We enhance the ABS language so that the programmer can specify a particular physical location where an object group is deployed.
- We experiment the ProActive backend on a realistic use case and show its benefits.

The paper is organized as follows. Section 2 reviews the background about ABS and ProActive and discusses motivations. Section 3 gives and explains the principles of the translation for each typical ABS statement. Section 4 presents an experimental evaluation of the ProActive backend and section 5 concludes.

2 Background

2.1 ABS

ABS [6] is an object-oriented language that targets modelling of distributed and concurrent applications. ABS supports concurrent object groups (COGs) and asynchronous method calls

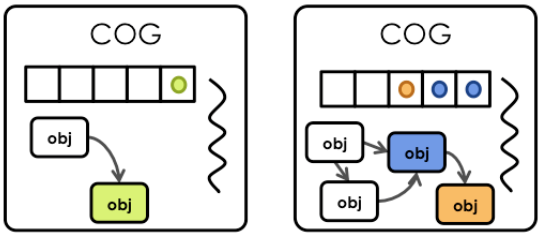


Figure 1: An example of ABS program execution with two cogs and six objects

with futures. Asynchronous method calls are distinguished from synchronous method calls via the operator `!`. Asynchronous method calls return future variables that are defined with the parametric type `Fut<T>`. The following example is an asynchronous call that returns a future of parametric type `V`:

```
Fut<V> future = object!method();
```

When a method is called asynchronously on an object, the call goes in the queue of the object's COG and becomes then a request. Thus, all asynchronous method calls are scheduled by COGs. Figure 1 represents an ABS program execution with 2 COGs and 6 objects. As displayed, a COG request queue holds requests that are meant for all objects it contains, mixed all together.

The requests are executed in a cooperative manner thanks to the `await` keyword. The `await` keyword can be used followed by a future or by a condition as in the two examples:

```
await future?;          // case #1
await a > 2 && b < 3;   // case #2
```

Those examples cause the execution thread to be released if (`case #1`) the future is not resolved or if (`case #2`) the condition not fulfilled. In those cases the execution thread is handed over to another ready request. A paused request is ready to resume when the future is resolved (in `case #1`) or when the condition is fulfilled (in `case #2`).

The `get` accessor can be used on a future variable to retrieve the value of the future, as follows:

```
V v = future.get;
```

The `get` accessor blocks until the future is resolved (and does not release the execution thread until then).

The Java backend of the ABS Tool Suite produce local Java programs that run in a single memory space. There is no current support for distribution, neither in the ABS language itself nor in any of the existing backend.

2.2 ASP/ProActive

ASP [2] is an active object language that enforces strict sequential execution of requests; active objects execute one request at a time and execution of requests cannot interleave. ProActive [3] is the Java implementation of ASP. In ProActive, active objects are transparent to the programmer: they follow as much as possible the syntax of regular Java objects. Method calls are implicitly asynchronous if the targeted object is an active object. The return type of asynchronous method calls is the same as the corresponding synchronous method calls: futures are not manipulated by the programmer. The execution blocks automatically when the value of a future is needed. Here is an example of a ProActive program:

```
T t = PAActiveObject.newActive(T.class, parameters, node);
V v = t.bar();
o.foo(v);
v.foobar();
```

In this example, an active object is created using the `newActive` ProActive primitive. Parameters of the `newActive` primitive typically include: the class to instanciate, the parameters of the constructor, and the node on which the active object should be deployed. The variable `v` is the result of the asynchronous call `bar` on `t`. The dynamic type of `v` is actually a future, that is a dynamically created subtype of `V`. Futures can be passed as parameters of further calls without blocking; they are first class futures. When a future value is needed to continue execution, such as in `v.foobar()`, a wait-by necessity automatically occurs until the future is resolved.

In ProActive, when an active object is created, there are in fact two different objects that are created:

- A local proxy. The reference given back by the `newActive` primitive is actually a reference to a proxy of the active object. As the active object is likely to lie on a different node, the proxy encapsulates all the needed code to reach the remote active object.
- The remote active object. This is the object the programmer think of manipulating when he manipulates the reference given back by the `newActive` primitive. Contrarily to the proxy, a single copy of the remote active object will ever be in the program for its whole lifetime.

ProActive is intended for distribution. It provides a mechanism to facilitate application deployment through the concept of virtual nodes: a virtual node is an aggregation of physical machines that are declared in configuration files (XML files). Such virtual nodes can be used in the code via their identifiers in order to choose a specific machine on which to deploy an active object.

2.3 MultiactiveObjects

Recently, an extension of the active object model has been proposed, the multiactive object model [8]. Multiactive objects leverage deadlock issues that are likely to arise when doing reentrant calls with active objects. The principle is to execute multiple requests in an active object in parallel, while controlling the concurrency thanks to the specifications of the programmer. Thus, multiactive objects are also better adapted to multicore architectures than active objects.

The multiactive object model has been implemented as an extension of ProActive. This extension comes as a small metalanguage to allow a programmer to declare which requests can safely be executed in parallel, namely which requests are *compatible*. This metalanguage is based on the Java annotation mechanism. A programmer can use these annotations on top of a class so that all objects of this class are automatically multiactive objects when instanciated with the `newActive` ProActive primitive[1].

Request compatibility can be specified following three steps:

- A programmer first uses the `@Group` annotation on top of a class to define a group of requests. A group is meant to gather requests that have the same concerns and/or the same compatibility requirements.
- Then, a programmer uses the `@MemberOf` annotation on top of a method definition to make this method belong to a particular group (previously defined).

[1] if there is no annotation, then objects crerated with the `newActive` primitive remain basic active objects

- Finally a programmer uses the `@Compatible` annotation to specify which group is compatible with which other group, so, in extension, to specify which requests can be run in parallel safely.

Below is an example of an annotated class `MyClass`. The `selfCompatible` parameter of the `@Group` annotation defines whether two different requests *of the same group* can safely run in parallel. The internal scheduling of a multiactive object automatically takes into account the

```
@DefineGroups({
  @Group(name="group1", selfCompatible=true),
  @Group(name="group2", selfCompatible=false)
})
@DefineRules({
  @Compatible({"group1", "group2"})
})
public class MyClass {
  @MemberOf("group1")
  public ... method1(...) { ... }
  @MemberOf("group2")
  public ... method2(...) { ... }
  ...
}
```

Figure 2: Example of annotated class

compatibility specifications of the programmer. The default scheduling policy that is applied is the following: a request is executed if it is compatible with the request that are already executing *and* compatible with the previous requests in the queue. The first condition ensures safety and the second condition ensures a maximum parallelism while avoiding starvation.

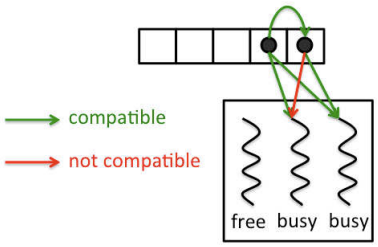


Figure 3: An example of internal checking for request scheduling

Recently, additional features have been added in multiactive objects to better control the scheduling of requests [9]. The new features include the possibility to set a limit on the number of threads that can be created inside a multiactive object. This can be done using an additional annotation on top a class, `@DefineThreadConfig`, like in the following:

```
@DefineThreadConfig(threadPoolSize=8, hardLimit=false)
```

The `threadPoolSize` parameter defines the maximum number of threads in a multiactive object. The `hardLimit` parameter defines the kind of limit that is applied: if it is true, it is the total number of threads that is limited by the `threadPoolSize` parameter. If it is false, only the number of active threads is limited by the `threadPoolSize` parameter. Active threads are threads that are currently executing a request and not blocked in a *wait-by-necessity* state. In consequence, the number of waiting threads is not limited when the `hardLimit` parameter is set to false. For example, suppose, considering the above `@DefineThreadConfig` annotation, that the eight threads execute a request each. If one of those threads stops because it waits

for a future to be resolved, then, because the `hardLimit` is set to false, a ninth thread will be created to execute another request. In this case, there still are eight active threads but one waiting thread in addition. Afterwards, when the request that waits for a future is ready to resumed it waits until a slot of active thread is available.

Another specific extended feature of multiactive object is the limitation of threads *per group*. The `@Group` annotation supports two additional parameters for this purpose, as follows:

```
@Group(name="routing", selfCompatible=true, minThreads=2, maxThreads=10)
```

The `minThreads` parameter reserves threads from the initial thread pool. In this example, it guarantees that the `routing` group will always have two threads devoted to its requests, so that at least two requests of the routing group can be run at any moment. Those reserved threads cannot be used by other groups. On the other hand, the `maxThreads` parameter defines the maximum number of threads that can be used at the same time by requests of this group. In the example, the `routing` group can have at maximum ten of its requests executed at the same time. The `minThreads` and `maxThreads` parameters can be seen as the lower and upper bound on the number of threads that a group holds at a given time.

Multiactive objects enable a high level implementation of precise scheduling policies. We use the introduced features to implement our ProActive backend. In the following, we use the term ProActive to denote the latest version of the software with multiactive object features, if not mentioned otherwise.

2.4 Positioning

Although ABS and ProActive are based on same foundation, namely the active object programming model, they are not meant for the same purpose. On one hand, ABS first objective is to provide a language to *model* distributed applications and to run the models in order to find design issues, or to show the suitability of a solution. On the other hand, the main goal of ProActive is to build *executable* distributed systems. However, in order to satisfy increasing expectations, ABS might have to support the distribution of its programs in the future, to be able to experiment in realistic conditions or to run very large applications. Instead of building a deployment mechanism from the beginning, ABS could benefit from the expertise of ProActive in this domain. However, as ABS was not initially designed for distribution, practical issues arise naturally.

One of the characteristics of ABS is that all objects are accessible from all other objects, regardless whether they are in the same COG or not. In consequence, the programmer always manipulates a *direct* reference to an object. When running an application in a distributed manner, an application-level index must be maintained in order to access objects that are not in the same memory space, and in general to access objects throughout the network. To understand the challenge, consider the following ABS program:

```
Server server = new cog Server();
Fut<Identifier> idFut = server!getId();
Identifier id = idFut.get;
```

This program executes in a COG[2], and creates a new COG with inside a new object, `server`. The server identifier is then retrieved through an asynchronous method call and further designated by the local variable `id`. According to the ABS semantics, `id` must be a direct reference to the `id` object. But as the `id` object is supposed to be in another memory space (since the cog should naturally be the unit of distribution), an indexing system must be set to remotely reference the `id` object. The problem is that it should be the same for all the other objects created

[2]An ABS piece of code is always executed in a COG, which can be the default one.

by `server`, or linked to `server`. In the end, all objects of the application should appear in the indexing system, as they should be referenced as if they were local. This doubtlessly leads to a scaling problem.

In contrast, in ProActive, a direct remote reference can only be a reference to an active object. For the other objects, a copy of the object is accessed instead. The reason for that is to keep the footprint of the global indexing system low. In order to get a scalable and effective ProActive backend, it is necessary to comply with this general rule. In consequence, the main challenge for the ProActive backend is to recreate the behavior *by reference* of ABS with the behavior *by copy* of ProActive.

3 From ABS to ProActive

ABS and ProActive have conceptual differences that need particular attention for the design of the ProActive backend. The main differences are in:

- the active object model: ABS is based on a component object group model whereas ProActive features active and passive objects.
- the implementation: ABS uses explicit asynchronous calls, and an explicit future type; ProActive handles both asynchronous calls and futures in a transparent way.
- the threading model: ABS follows a cooperative scheduling policy; ProActive does not provide a way to pause a request execution in favor of another request. However, ProActive provides *controlled multithreading*.

Those differences are challenging in the objective to translate from one language to the other without breaking the initial semantics. In this section, we present the global principle of the ProActive backend and we detail the translation of ABS programs into ProActive programs, considering the three difficulties we have just introduced.

3.1 Object Instanciation

ABS and ProActive are both object-oriented languages that provide active objects. However, they are based on different active object models. We distinguish three different active object models, listed hereafter:

Uniform Object Model. In the uniform active object model, all objects are active objects. All objects have their own request queue and their own private execution thread. In consequence, all communications between objects occur by posting a request to an object. This is the object model of Creol [4].

Non Uniform Object Model. Alternatively, active objects can get along with passive objects[3]. Two kind of objects then coexist, like in ProActive. A passive object cannot be directly accessed remotely; it has to be manipulated through the unique active object to which it is related, otherwise it is a copy of this object that is manipulated.

Object Group Model. The object group model gathers several objects so that they share the same request queue and execution thread, like in ABS and JCoBox [5]. In the ABS case, only one kind of object exists and all objects are directly accessible from any group.

[3]Passive objects are regular objects. They have no request queue, no dedicated execution thread and support only synchronous method calls.

The active object model drives the object creation process. To handle the difference between the object models of ABS and ProActive, in our ProActive backend we need to define what happens in ProActive when a new ABS object is created. In particular, we need to define the ProActive code that would be equivalent to the ABS new cog statement.

As in ABS objects should be accessible from all others, one could think that implementing all objects with a ProActive active object would meet the requirement. However, in practice, this is hardly manageable: a ProActive active object is associated a plain Java thread (not a logical thread). Thus, this solution would inevitably lead to a substantial context switch overhead, as the number of objects is large in the application, harming its overall performance. In our ProActive backend, only the COGs are active objects: they have a private request queue and execution thread. The reasons for this choice are the following. First, it is natural to consider the COG as the unit of distribution, and ProActive active objects are the unit of distribution. Second, when a ProActive active object is created, it is automatically put in a registry that is accessible through the network[4]. A COG object is thus accessible network-wide via an URL. Thanks to this registry, a COG object can be the entry point to all the objects it contains. All objects other than COG objects can then remain passive, thus preserving the performance of the ProActive backend. Thus, we have a two-level hierarchical indexing of objects:

- A first level of network-wide accessible COG objects. This mechanism is integrated in ProActive as it is based on RMI [10].
- A second level of locally-accessible objects. This mechanism is implemented in the COG class using a map from object identifiers to object references.

This classification implies that the application always manipulates local references except for COG objects that can be manipulated through remote references. To actually apply this strategy, we need to implement in ProActive the translation of the `new cog` statement. Consider the following ABS line of code:

```
Server server = new cog Server();
```

We translate this single line of code into three ProActive lines of code:

```
Server server = new Server();
COG cog = PAActiveObject.newActive(Cog.class, new Object[]{
                absRuntime, instanciatingClass, gcm}, gcm.getANode());
cog.registerObject(server);
```

The first line creates a regular server object. The second line uses the `newActive` primitive of ProActive to create a new COG active object. Several parameters are passed to the `newActive` primitive to (i) ensure that the new COG object is well-constructed and to (ii) specify onto which node it should be deployed (we will go back to this point later on). The third line is paramount: as the new COG was created remotely on another node, we need to send over the newly created server object onto the same node. Indeed, an object and its COG must always be located on the same memory space, since a COG should access its objects through local references. To send over the server object to the memory space of its COG, we use the `registerObject` method of the COG, giving `server` as parameter. As the reference to the COG is a remote reference, the `registerObject` call is a remote call. Consequently, the parameter of this call, the server object, will be copied away according to the ProActive object model, and all the objects created along with the server object will be copied too, if any. Then of course, to comply with the initial ABS program, all further method calls on `server` should be run on the remote server object, and not on the server object that was created locally. Nevertheless, we still need the local server object to be able to get back to the targeted remote server object, as we will see in the next section.

[4]This is in fact the RMI registry, as ProActive is based on RMI

3.2 Asynchronous Method Calls

So far, object creation in the ProActive backend is compliant with the ABS active object model. But ABS and ProActive still have different ways to process asynchronous method calls. In particular, ABS features explicit asynchronous method calls with explicit futures, whereas ProActive manages them transparently. In the initial Java backend, a synchronous method call was distinguished from an asynchronous method call by calling a different method in each case. In our ProActive backend, we simply removed this distinction, as method calls are implicitly asynchronous on active objects.

However, in our ProActive backend, only COG objects are active objects, which means that they are the only objects on which asynchronous method call can be run. Thus it is not technically possible to directly run an asynchronous method call on a object that belongs to a different COG, as it is done in ABS. We have designed an adapted strategy to pass by this limitation. Consider this ABS piece of code:

```
server!start();
```

We suppose that `server` is a the server object created in a new COG of the previous example. The idea to translate this asynchronous method call in ProActive, is to run an asynchronous method call on the COG of `server` and then let this method retrieve the server object and run the desired method on it. Here is the translation the ProActive backend provides:

```
server.getCog().execute("start", new ABSType(){}, server.getId());
```

More precisely, we first retrieve a reference to the COG of the server object by using the local reference of the server object. Note that this server object is not the one on which we actually want to run the `start` method. However, this server object has the right remote reference to its COG (since in ProActive, active object references are unique) and it is this remote reference we get when calling the `getCog()` method[5]. Then, the `execute` method is called on the COG object returned by `getCog()`, and this method call is implicitly asynchronous by the nature of the COG object. The `execute` method has been added in the COG class in order to execute all the asynchronous method calls of the program. The call to this method drops an `execute` request in the request queue of the targeted COG. When the `execute` request is run, it uses the third parameter of the request, `server.getId()`, to lookup the local reference of the server object that is in the memory space of the targeted COG and run the `start` method synchronously on it by reflection. The first parameter of the `execute` method is the name of the method to run; the second parameter of the `execute` method is an array of the parameters of the method to run, all under the `ABSType` type.

Several things can be noted about the `execute` methods. First, as it is a ProActive asynchronous remote call, all of its parameters are copied away to be able to correctly run the method in the remote place. For the first and third parameters, the fact that they are copied does not matter because they are immutable variables, so having many independent copies of them does not change the behavior of the program. However, for the second parameter of the `execute` method (that holds the parameters of the call to make), the correctness of having a copy of it might not be obvious. In the previous example, the `start` method had no parameters, this is why the `ABSType` array was empty. But consider another asynchronous method call like this one:

```
server!start(p1, p2);
```

This asynchronous method call has two parameters, `p1` and `p2`. This is translated through the ProActive backend this way:

[5] Actually, this returns the local reference of a proxy to the remote COG, which is equivalent to a remote reference.

```
server.getCog().execute("start", new Object(){p1, p2}, server.getId());
```

When running the `execute` method, objects `p1` and `p2` are copied to the memory space of the targeted COG. Thus, two versions of `p1` and `p2` exist in the system at runtime, and the copies of `p1` and `p2` might not reflect the latest version of `p1` and `p2` when the `start` method is executed. In ABS, parameters `p1` and `p2` are manipulated by reference in the `start` method, and not by copy as it is the case in ProActive. However, in practice, the `start` method can only manipulate `p1` and `p2` through asynchronous method calls if they are not from the local COG[6]. In consequence, any method call on the copies of `p1` or `p2` is actually delivered first, as explained before, to the COG of the initial `p1` and `p2`. So in fact, any manipulation of those copies always gets back to their initial version by global referencing, thus reflecting all modifications that occurred meanwhile. Thanks to this mechanism, we are able to reproduce the behavior by reference of ABS using the behavior by copy of ProActive.

In the end, when we copy an object from one node to another, we only need two things: its identifier, to be able to retrieve the local object in the right memory space, and a reference to its COG. All the others attributes of the objects can actually be omitted since it will not be used on the copy side. This observation allows us to optimize object copy to reduce at minimum the amount of information that is sent to a remote node. We have thus few data transfers but in return, lots of communication.

3.3 Threading Models in Active Objects

Another fundamental difference between ABS and ProActive is in their threading model. We distinguish three different threading models in active object languages.

Single-threaded. Within an active object, requests are executed sequentially without interleaving possibilities. This is the threading model of ProActive active objects, if no annotation for multiactivity is found.

Cooperative. A running request can explicitly release the execution thread to let another request run. Requests are not processed in parallel but they might interleave. Creol, JCoBox, and ABS comply to this threading model.

Multi-threaded. Within an active object, requests are executed in parallel using different threads, without pausing nor yielding. This is the threading model of ProActive with multiactive objects.

Compared to ABS, ProActive does not offer the possibility to pause a request for the benefit of another request. To get a similar behavior in ProActive, we use the possibilities of multiactive objects.

Multiactive objects provide several mechanisms to control the internal scheduling, such as compatibilities and thread limits, as seen in Section 2.3. By mean of tuning those mechanisms very carefully, we can get a precise scheduling policy. In ABS, the cooperative threading model is materialized by the `await` statement. Therefore, in the ProActive backend we modify the translation of the `await` statement to make it interfere with the internal scheduling of multiactive objects. To apply our strategy, we need to modify the way the `await` and `get` ABS statements are translated in the Java backend.

[6]In ABS, synchronous method calls are enabled only for objects that are in the same COG

3.3.1 `await` statement on futures

The `await` statement followed by a future should yield if the future is not resolved at time of the `await` evaluation. In ProActive, a special primitive is provided to interact with a future. We use this primitive as a basis of the `await` translation. Consider this ABS program:

```
Fut<Bool> ready = server!start(); (1)
await ready?;                     (2)
```

We translate the second line of code `(2)` in the following ProActive line of code:

```
PAFuture.getFutureValue(ready);   (2)
```

The `getFutureValue` primitive of ProActive is meant to explicitly wait for a future value to be resolved, provided that the given variable is actually a future. Note that, conceptually, any method invocation on the future would trigger the *wait-by-necessity* mechanism, as we need here, but for the code to be cleaner we prefer relying on the ProActive API. In our case, the issue with this call is that it is blocking. With no further configuration, this would be like doing an ABS `get` statement: no other request would be executed in this COG until the future is resolved. As we have seen in the section 3.1, the COG object receives all the asynchronous method calls of the objects it contains, and run them by reflection through the `execute` method. In consequence, all the asynchronous method calls of the entire program are executed through one method, so if we control this method, we control the way all the asynchronous methods are executed. Thus, in order to achieve the desired request scheduling, we configure the COG class and its `execute` method with multiactive object annotations in three steps:

1. We create a particular group of requests with the `@Group` annotation, named `scheduling`. We declare this group `selfCompatible`, so that several requests of this group are allowed to execute in parallel.

2. We assign the `execute` method to the `scheduling` group (`@MemberOf` annotation).

3. We declare with the `@DefineThreadConfig` that an instance of COG has a thread limit of 1 (`threadPoolSize = 1`), and that this limit is not a hard limit (`hardLimit = false`). This means that only *active* threads are counted in the thread limit of 1; the threads that are waiting for the future are then not counted in this limit.

Hereafter is the big picture of the COG class with the previous customization. Considering the

```
@DefineGroups({
  @Group(name="scheduling", selfCompatible=true)
})
@DefineThreadConfig(threadPoolSize=1, hardLimit=false)
public class COG {
  ...
  @MemberOf("scheduling")
  public ABSValue execute(UUID objectID, String methodName, ABSType[] args) {
  ...
  }
  ...
}
```

above definition, what happens now when the program encounters the previous line of code `(2)` (`PAFuture.getFutureValue(ready)`) is that the current thread is put in the waiting state, if the future `ready` is not resolved at this point. This thread is now a waiting thread, and because the global thread limit only counts active threads, another thread can be created or resumed since it is not counted in the global thread limit any more. The global thread limit

being equal to 1, as we configured, there can be only one single active thread at a time, which is exactly the ABS behavior.

Altogether, thanks to a few annotations in the COG class, we are able to make the ProActive `getFutureValue` primitive work like the `await` statement of ABS, even if they had a different behavior initially.

3.3.2 `get` statement

The translation of the get statement would have been straightforward if we had not configured the COG class for the translation of the `await` statement. Indeed, by doing so, we have disabled the blocking aspect of the ProActive `getFutureValue` primitive. The `get` statement should block the execution thread without handing it off to another request. So what we need to translate the `get` statement is to temporarily go back to the initial behavior of the ProActive `getFutureValue` primitive (without annotations). In the `await` case, what causes another request to execute is the setting of the `hardLimit` parameter of the COG class to false, which releases the limit so that it counts only active thread. Therefore, to translate the `get` statement, we set the `hardLimit` parameter to true just for the time the future is waited. This ensures that no new thread is started, and that no waiting thread is resumed, since the global limit of 1 is already reached, at least counting the current thread. When the current future is resolved, setting back the `hardLimit` parameter to false ensures that execution resumes normally. Consider this new ABS program:

```
Fut<Bool> readyFut = server!start(); (1)
Bool ready = readyFut.get;           (2)
```

We translate the second line of code `(2)` in the following ones:

```
getCOG().switchHardLimit(true);
PAFuture.getFutureValue(ready);
getCOG().switchHardLimit(false);
```

We first set the `hardLimit` of the current COG to true thanks to the `swichHardLimit` method call. Note that this call is synchronous since the retrieved COG is the current one. This way, we are sure that the limit is switched when executing the next line of code. Then, the future `ready` is waited. The next `switchHardLimit` call can only be executed when the future is resolved, thanks to the *wait-by-necessity*, and its purpose is to reset the `hardLimit` to false, to restore the previous behavior.

So thanks to the limit type switch, this translation offers the same behavior as the ABS `get` statement.

3.3.3 `await` statement on conditions

Another usage of the ABS `await` statement is to use it followed by a condition. For this usage of the `await` statement, we cannot rely on the ProActive `getFutureValue` primitive at first glance, because a condition is obviously not a future variable. However, we can still wrap the condition evaluation in a method call, possibly returning a future value and waiting for it in a `getFutureValue` call.

During compilation, when such an `await` statement is encountered, we generate a method that lively checks if the condition is true. Then, we translate the statement by an asynchronous call to a special method in the COG class that we have added, with signature `awaitCondition(objectID, method, parameters[])`. We call the `awaitCondition` method on the current COG, giving it as parameter the current object and the generated method corresponding to the condition, and the condition members as parameters if they are not globally defined. Again, we wrap this

call in the `getFutureValue` primitive, in order to wait for the result of this asynchronous call, i.e. when the condition evaluates to true. For example, when the following ABS code is encountered:

```
await a == 3;
```

First, a corresponding method is generated[7]:

```
public void cond7517d1ff7c52(int x) {
  while(x != 3) {
    Thread.sleep(500);
  }
}
```

Then, the ABS `await` statement is translated into:

```
PAFuture.getFutureValue(
    this.getCog().awaitCondition(this.getId(), "cond7517d1ff7c52", a));
```

The `awaitCondition` call executes by reflection the method `cond7517d1ff7c52` on the object retrieved with the given identifier, passing the parameter `a`. The reason why we need to rely on an additional method to perform the lively waiting is because we need to release the execution thread slot to let other request progress meanwhile. But without any further configuration, this strategy potentially leads to a deadlock when the generated method gets executed in the COG. Indeed in this case, the `awaitCondition` method execution could take up the single execution thread and never release it because, as no other request could progress, the condition would never evaluate to true. In order to prevent the condition method from monopolizing the execution thread, we dedicate some new threads to the execution of condition methods only.

We create a new group of requests, named `waiting`, with the `@Group` annotation, and we put the `awaitCondition` method in this group, thanks to the `@MemberOf` annotation. We also declare the `waiting` group `selfCompatible`, because several `awaitCondition` requests could evaluate conditions at the same time. To be able to execute the `awaitCondition` requests in parallel, we arbitrary increase the global limit size by 50 threads, to have at maximum 50 concurrent condition evaluations in one COG, plus 1 thread for processing the standard requests (from the `scheduling` group). As waiting conditions and standard requests should execute in parallel, we declare the `scheduling` and `waiting` group compatible. Nevertheless, we still must prevent the 50 new threads from executing requests of the `scheduling` group, because otherwise it would break the ABS cooperative scheduling. For that, we specify a minimum and a maximum of 50 threads for the `waiting` group in its declaration. The result of this specification is that, not only the waiting group will be able to use up to 50 threads, but also those 50 threads cannot be used by other groups. So in fact we have a clear separation between the single thread used by the `scheduling` group, and the threads used by the `waiting` group. The COG class is thus augmented with new annotations, as follows: This solution is fully compliant with the ABS semantic and proves the power of multiactive object annotations. Note that, although effective, this solution could be enhanced with a callback when the condition is true, instead of having a live waiting.

3.4 Distribution

In the previous sections, we have presented the core translation of ABS programs into ProActive ones. However, correctly translating the concepts of the language is not the only issue in our objective: we still have to handle the challenge of distribution. Even though ProActive embeds

[7] The code is lighten for clarity reasons.

```
@DefineGroups({
  @Group(name="scheduling", selfCompatible=true),
  @Group(name="waiting", selfCompatible=true, minThreads=50, maxThreads=50)
})
@DefineRules({
    @Compatible({"scheduling", "waiting"}),
})
@DefineThreadConfig(threadPoolSize=51, hardLimit=false)
public class COG {
  ...
  @MemberOf("scheduling")
  public ABSValue execute(UUID objectID, String methodName, ABSType[] args) {
  ...
  }
  @MemberOf("waiting")
  public ABSType awaitCondition(
      UUID objectID, String methodName, ABSType[] args, Class<?>[] args) {
    ...
  }
  ...
}
```

all that is needed for distribution, we still had to make minor modifications on the ABS frontend to satisfy the requirements of distribution.

- *Serialization.* The main challenge when moving objects from one memory space to another is to represent the object in a way that is conveyable throughout the network, that is, how to serialize the object so that it can be reconstructed when arrived at destination. As ProActive is based on RMI, all objects that are part of a remote method call (parameters and return values) should implement the Java Serializable interface for this purpose. We thus modified in consequence some classes of the ABS frontend to make them implement this interface.

- *Copy Optimization.* In order to minimize copy overhead, we declare during compilation as many attributes as possible with the `transient` Java keyword, which prevents those attributes to be copied when the containing object is sent away. This is possible because, in our implementation, we only need a single version of an object that contains all the attributes. Indeed, any manipulation of an object is made on this initial version, as seen in Section 3.2. In our case, the only attributes we need to copy over are the identifier of the object and the remote reference to its COG. Everything else can be declared with the `transient` keyword.

- *Deployment.* When a programmer intends to distribute a program he needs a way to mention the set of machines he wants to use and to specify which part of code should be executed on which machine (or a particular policy must decide it, but the chosen policy still has to be specified). ProActive embeds a deployment descriptor, based on XML configuration files. In those configuration files, the programmer specifies to which physical machine corresponds each node. We slightly modified the ABS language in order to offer the possibility to designate a node on which a COG must be deployed, linking it to the deployment descriptor. The `new cog` statement can be now optionally followed by a string that identifies a groups of machines, as known as node, like for instance in:

  ```
  Server server = new cog "mynode" Server();
  ```

  Such kind of `new cog` statement is translated by first retrieving the specified node thanks to its identifier using the deployment descriptor API of ProActive, and by secondly giving it as a parameter of the ProActive `newActive` primitive. This causes the new COG to be

created on a machine that belongs to the specified node, following a particular retrieving policy[8]. The main deployment descriptor file of the previous example can look like the following:

```
<GCMDeployment>
  <hosts id="mynode" hostCapacity="1"/>
  <sshGroup hostList="machine1 machine2 machine3"/>
</GCMDeployment>
```

This file defines the network name of the machines that are part of the node, that is `machine1`, `machine2`, et `machine3`. Many other deployment options can also be defined, as specified in GCM. In the case the `new cog` statement is used without node specification, the COG active object is simply created on the local node.

4 Experimental Evaluation

In order to validate the benefits of our approach, we conducted an experimental evaluation of the ProActive backend and we compared it with the performance of the initial Java backend. The considered use case is the pattern matching of a DNA sequence against a DNA database using the MapReduce programming model [11]. We implement the MapReduce programming model in ABS with a MapReduce class that is responsible for splitting the input data and dispatching `map` and `reduce` calls to several workers (class Worker). Each worker is instanciated in a new COG so that workers can work in parallel. Workers do not communicate with each other, they communicate with the MapReduce object only.

We consider a searched pattern of 250 bytes, and a database of 5 MB of DNA sequences. The MapReduce object divides the database into 100 equal parts (50 kB each), and creates 100 pairs `(pattern, part)`. The pairs are distributed evenly to the workers and processed in parallel through the map method. Each map tries to match the pattern on the given sample, and returns the maximum matching sequence found in this sample. Once all mappers are finished, all the results that are local to a map are aggregated and passed to a single reducer that outputs the global maximum matching sequence of the pattern against the whole database. The reduce phase is negligible compared to the map phase. Indeed, the map phase is based on an exponential algorithm, showed in Algorithm 1, whereas the reduce phase just has to compute the maximum of 100 entries. We computed the execution time of the whole use case when varying the number

Algorithm 1 Maximum matching sequence algorithm - map phase

```
while pattern not ended do
    while sample not ended do
        while pattern matching sample do
            update maximum matching
        end while
    end while
end while
```

of workers, i.e. when varying the degree of parallelism. When using the initial Java backend, we run the use case on one single machine, since it does not support distribution. When using the ProActive backend, we deploy two workers (two COGs) per machine, to fully exploit the machine hardware. For all experiments, we used the machines of a single cluster of the Grid5000 platform [12]. All the machines have 2 CPUs of 2.6 GHz with 2 cores each, and 8 GB of RAM.

[8]The default policy evenly distributes the active objects.

Figure 4 shows the results when using from 2 to 50 workers, therefore using from 1 to 25 physical machines in the case of the ProActive backend ("Distributed" line on the figure).

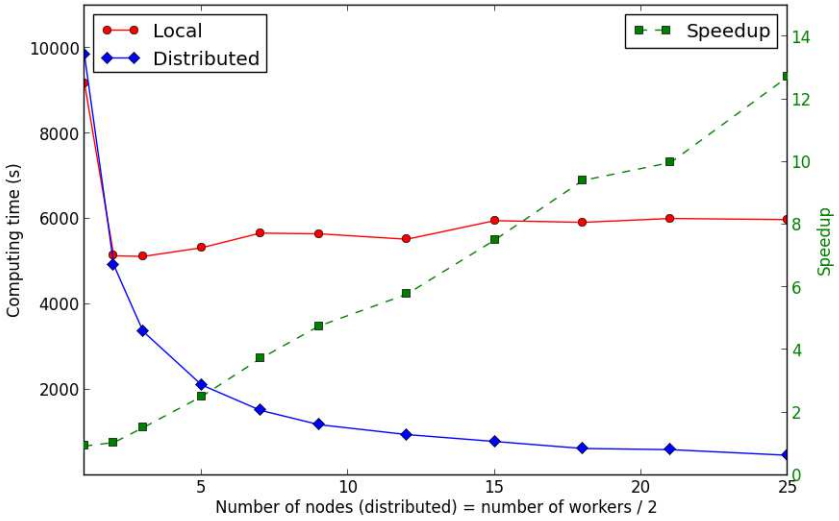


Figure 4: Execution time for local and distributed execution per number of nodes/workers. Each measurement dot is an average of five executions.

The execution time of the use case with the ProActive backend decreases sharply for the ten first added machines and then continuously decreases slowly after using more than a tenth of machines. On the other hand, the initial Java backend has an optimal degree of parallelism of 4 workers, which actually is the number of computing units of the machine. Then, when the number of workers increases, the execution time increases slowly as well, whereas the degree of parallelism is augmented. This raise is due to an increasing context switching overhead, as all the threads are executing on the same machine. On the opposite, increasing the degree of parallelism when using the ProActive backend is always gainful, even though the gain tends to decrease as the number of workers (and of machines) increases. In summary, the speedup achieved from the initial Java backend to the ProActive backend grows linearly for the tested values. In the best case, the code produced by the ProActive backend runs 12 times faster than on 25 machines than the code produced by the initial Java backend on a single machine. More reasonably, we can point out that simply using 10 machines makes the use case complete 5 times faster, precisely in 19 minutes instead of 1 hour and 35 minutes.

Nevertheless, we can also notice that, at the lowest number of workers depicted, the regular Java backend performs better than the ProActive backend: it is 7% faster. This is due to the overhead induced by the distribution: distributing involves more computing units but costs more in terms of serialization and communication. Thus, it is useless to use the ProActive backend for a small number of workers, because the parallelism achieved in shared memory is more efficient in this case. This shows that the ProActive backend should be used only when the use case includes computationally intensive work, that can be split into subsets, to distribute the computational load across several machines. However, the ProActive backend is small enough to be beneficial quickly, as the size of the use case grows.

5 Conclusion

In this paper, we presented a new backend for the ABS modelling language. This backend is based on the ProActive Java library and has the particularity to enable distribution of ABS programs across a set of machines without major changes for the ABS users. In addition to describing the tool, we explained how we designed the translation between the two active objects language underlying, ABS and ProActive. We informally showed that we could simulate the behavior of ABS COGs with ProActive multiactive objects, providing a convincing translation of asynchronous calls, and of `new cog`, `await`, and `get` ABS statement. We also took care of practical issues of distribution, and optimized the communications between objects. We finally presented an experimental evaluation that showed the benefits of the ProActive backend, that proved a speed up of more than 12 in the best case, compared to the initial Java backend.

A deeper outcome of this work is the comparison of the two active object languages, with their advantages and drawbacks. The object model of ABS is convenient to reason on it because there is no problem of locality, but it is in return hardly implementable in a distributed fashion as it is. On the other hand, the object model of ProActive natively integrates the notion of distribution but since each active object is associated a plain thread, it is harming to make an extensive use of active objects. ProActive encapsulates better the state of an active object than ABS though, by forbidding direct remote references to passive objects. Overall, the main limitation of our approach is that the initial ABS program should be relatively aware of the ProActive active object model, in order to balance the number of COG per machine and the number of objects in a COG in way that comply with this model.

References

[1] Lavender, R.G., Schmidt, D.C.: Active object: an object behavioral pattern for concurrent programming. In: Pattern languages of program design 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1996)

[2] Caromel, D., Henrio, L., Serpette, B.: Asynchronous sequential processes. Information and Computation (2009)

[3] : The Proactive middleware `proactive.inria.fr`.

[4] Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science (2006)

[5] Schäfer, J., Poetzsch-Heffter, A.: Jcobox: Generalizing active objects to concurrent components. ECOOP 2010 – Object-Oriented Programming (2010)

[6] Johnsen, E., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: Abs: A core language for abstract behavioral specification. In Aichernig, B., Boer, F., Bonsangue, M., eds.: Formal Methods for Components and Objects. LNCS. Springer Berlin Heidelberg (2012)

[7] : The ABS Tool Suite `http://tools.hats-project.eu/`.

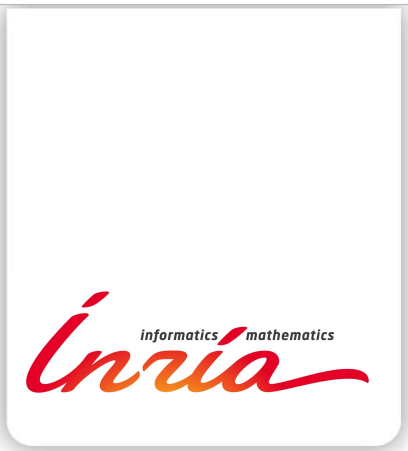
[8] Henrio, L., Huet, F., István, Z.: Multi-threaded active objects. In Julien, C., De Nicola, R., eds.: COORDINATION'13. LNCS, Springer (June 2013)

[9] Henrio, L., Rochas, J.: Declarative Scheduling for Active Objects. In Shin, S.Y., ed.: SAC 2014 - 29th Symposium On Applied Computing, Gyeongju, Corée, République De, ACM Special Interest Group on Applied Computing, ACM (March 2014) 1–6

[10] Wollrath, A., Riggs, R., Waldo, J.: A distributed object model for the javatm system. In: Proceedings of the 2Nd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2. COOTS'96, Berkeley, CA, USA, USENIX Association (1996) 17–17

[11] Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51**(1) (January 2008) 107–113

[12] : The Grid5000 platform `www.grid5000.fr`.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399